공학석사 학위논문

# 자바스크립트 엔진에서의 최적화된 기계어 코드 재활용

# Recycling the Optimized Machine Codes Generated by JavaScript Engine

2017 년 7 월

서울대학교 대학원

전기정보공학부

김 성 국

# 자바스크립트 엔진에서의 최적화된 기계어 코드의 재활용

지도 교수 문 수 묵

이 논문을 공학석사 학위논문으로 제출함
2017 년 7 월

서울대학교 대학원
전기정보공학부
김 성 국

김성국의 공학석사 학위논문을 인준함
2017 년 6 월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 이 혁 재 (인)

# 초 록

자바스크립트의 역할이 계속해서 증대되고 있으며, 자바스크립트 엔진의 성능이 중요한 이슈가 되고 있다. 본 논문은 자바스크립트 엔진의 성능을 향상하기 위한 방법으로 다단계(multi-tiered) JIT 컴파일 엔진에서 최적화 JIT 컴파일러가 생성하는 기계어 코드를 재활용하는 방식을 제안한다. 이 방식에서는 최적화 JIT 컴파일러가 생성하는 기계어 코드를 파일에 저장하였다가 다음 프로그램 수행에서 이를 재사용한다. 그리하여 엔진에서 컴파일 오버헤드를 제거하고 시작부터 양질의 코드를 사용하게 만든다.

기계어 코드를 재사용하기 위해서는 코드 내의 포인터들을 현재 수행에 맞게 패치 해주어야 하는데, 이를 위하여 기계어 코드와 함께 코드 내의 포인터 위치 및 포인터가 가리키는 객체의 정체를 같이 저장하는 테이블을 생성, 저장한다. 최적화 JIT 컴파일러에서는 베이스라인 JIT 에서와는 달리 완성된 테이블을 만들기 위해 최적화 과정 중에서 파생적으로 생성되는 포인터와 프로파일에 의해 고정 생성되는 포인터들까지 찾아서 처리해야 한다. 전자는 최적화 과정을 같이 따라가면서, 후자는 컴파일러가 포인터를 고정 생성하지 못하도록 바꾸어 처리한다. 이를 실제 자바스크립트 엔진 중의 하나인 JavaScriptCore 엔진에서 적용한 결과, 사용자가 선택적으로 이득이 되는 기계어 코드들만 재사용 했을 시 최대 29%, 평균 11%의 성능 향상이 있었다.

**주요어 : 자바스크립트, 가상 머신, JIT 컴파일러, 컴파일 오버헤드, 포인터, 기계어 코드**
**학 번 : 2015-22777**

# 목 차

# 그림 목차

# 제 1 장 서 론

많은 서비스들이 계속해서 웹 플랫폼으로 몰리고 있다. 웹 플랫폼을 이용한 서비스들은 기존 프로그램들과는 다르게 별도의 설치 작업을 필요로 하지 않는다. 자바스크립트는 웹 플랫폼에서 요소들의 동작을 다루는 표준 언어로, 웹 플랫폼의 인기에 힘 입어 가장 인기 있는 언어 중 하나로 성장하였다. 현재는 클라이언트뿐만이 아니라 Node.js라는 서버 애플리케이션 플랫폼을 통해서 서버 쪽 프로그래밍에서도 널리 이용되고 있다. 심지어 MEAN(MongoDB, Express, AngularJS, Node.js) 스택[1]이라는 웹 개발 기술 스택은 자바스크립트 만으로 모든 웹 서비스를 작성 가능하게 해준다. 이처럼 자바스크립트는 다양한 환경에서 널리 사용되고 있고 자바스크립트 프로그램을 실행시켜주는 자바스크립트 엔진의 성능이 점점 중요해지고 있다.

이러한 자바스크립트 엔진의 성능을 향상시키기 위하여 그 동안 다양한 기법들이 시도 되어 왔다. 그 중에서도, JIT 컴파일(JITC, Just-in-Time Compilation) 기법[2]이 엔진의 성능을 향상시키는 주된 방법으로 자리잡고 있다. JIT 컴파일을 하는 엔진들은 프로그램을 실행 하는 도중에 자바스크립트 프로그램내의 함수들을 기계어 코드로 컴파일을 한다. 컴파일 과정에서 당연히 비용이 발생하지만, 기계어 코드를 실행하는 것이 바이트 코드를 하나하나 해석하는 것보다 빠르기 때문에 전체적으로는 실행 시간이 단축 된다. 하지만 여전히 컴파일 과정의 오버헤드가 실행 시간에 그대로 포함되고, 컴파일 되기 전까지는 함수가 느린 방식으로 실행되어야 한다. Sunspider 벤치마크[3]를 대상으로 분석해본 결과, 최적화 JIT 컴파일러의 컴파일 시간은 전체 실행 시간의 약 20% 나 차지할 만큼 오버헤드가 상당하다는 것을 확인할 수 있었다(단일 thread로 컴파일).

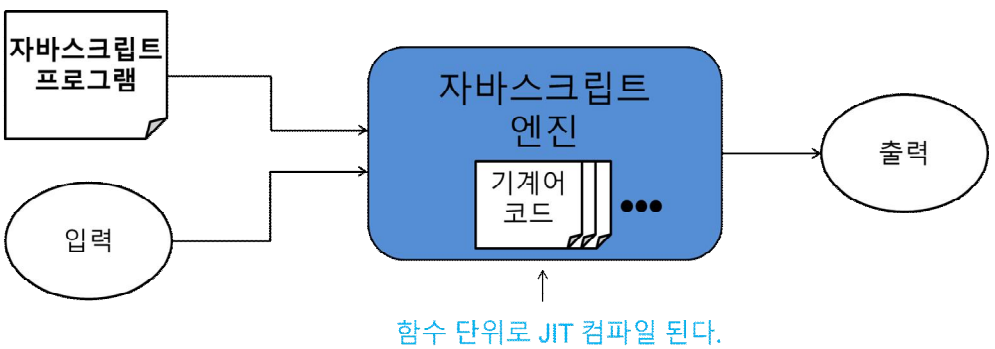
이러한 단점을 개선하기 위하여 JIT 컴파일 기법의 대안으로

AOT 컴파일(AOTC, Ahead-of-Time Compilation) 기법[4, 5, 6, 7]이 제안되었다. 본 논문에서 서술하는 AOT 컴파일 기법은 이전 실행의 JIT 컴파일러가 생성하는 기계어 코드들을 파일에 저장하였다가 다음 프로그램 실행에서 재활용하는 방식을 의미한다. 이 기법을 통해 자바스크립트 엔진은 함수에 대한 컴파일 과정을 실행마다 반복하지 않게 된다. 기존 자바스크립트 엔진의 AOT 컴파일 기법을 다룬 연구[5, 7]에서는 최적화가 거의 적용되지 않는, 베이스라인 JIT 컴파일러를 대상으로 진행됐었다. 이번 연구에서는 여기서 한 단계 더 나아가 최적화를 적용하는 JIT 컴파일러를 대상으로 AOT 기법을 적용한다. 이를 통해 컴파일 오버헤드를 제거할 뿐만 아니라, 시작부터 더 양질의 기계어 코드로 프로그램을 실행하게 된다. 자바스크립트 엔진에서 최적화를 하는 JIT 컴파일러들은 언어의 특성상 런타임 관찰 결과들에 의존할 수 밖에 없고, 이는 최적화 과정들과 더불어서 베이스라인 JIT 컴파일러에는 없었던 새로운 문제들을 제기한다. 본 논문에서는 이 아이디어를 WebKit의 JavaScriptCore[8]라는 자바스크립트 엔진에 실제로 구현하며 겪었던 문제들에 대해서 설명한다. 기존 자바스크립트 엔진의 구조, 제안하는 최적화된 기계어 코드의 재활용 방식, 실험 결과, 결론 순으로 설명하도록 하겠다.

# 제 2 장 자바스크립트 엔진의 구조

## 제 1 절 자바스크립트 엔진

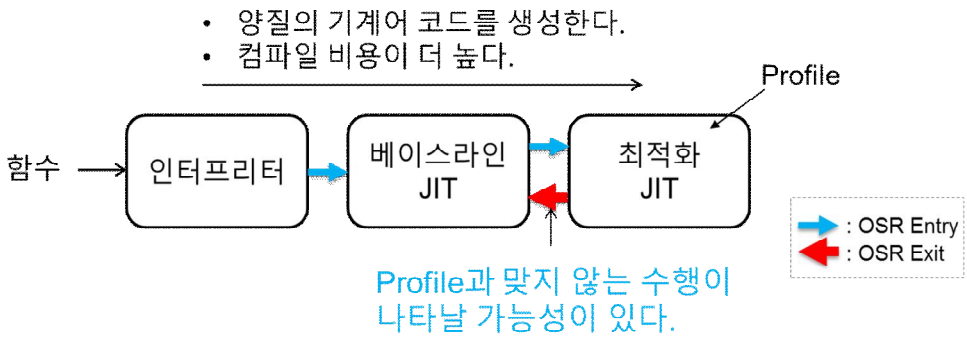
자바스크립트 엔진이란 자바스크립트로 작성된 프로그램과 그 프로그램의 입력을 받아 실행을 하며 출력을 내는 프로그램을 의미한다 (그림 1 참조). 자바스크립트 엔진은 실행 도중 프로그램의 함수들을 컴파일 할 수 있도록 JIT 컴파일러를 탑재하고 있다. 엔진은 자주 실행되는 함수들을 더 빠르게 실행하기 위하여 런타임 도중에 컴파일 한다.




[그림 1] 자바스크립트 엔진

엔진에서 JIT 컴파일을 하는 부분만을 더 자세히 살펴보면 그림 2와 같은 구조로 구성되어있음을 확인할 수 있다. 이는 다단계(Multi-tiered) JIT 컴파일(JITC, Just-in-Time Compilation) 기법을 사용하는 구조이다. 자바스크립트 엔진들은 보통 하나의 JIT 컴파일러가 아니라 최적화 수준에 따라 여러 개의 JIT 컴파일러를 갖는다. 그림 2에서 오른쪽으로 갈수록 더 높은 실행 단계를 의미하는데, 높은 단계일수록 더 양질의 최적화된 코드를 생성하는 JIT 컴파일러를 사용한다. 컴파일 결과물의 질과 컴파일 비용 사이의 트레이드오프 관계에 따라서 높은 단계일수록 컴파일 비용도 커진다. 따라서 모든 함수들은 맨 처음에는 가

장 낮은 단계인 인터프리터 단계에서부터 시작을 하고 실행 빈도가 높아짐에 따라 위의 단계로 올라간다. 위의 단계로 올라가는 것을 OSR Entry라고 부른다.



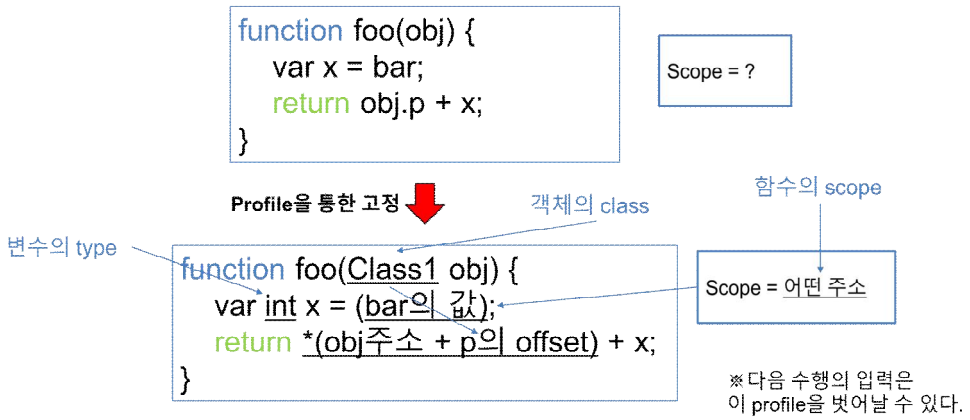

[그림 2] 자바스크립트 엔진의 다단계 JIT 컴파일 방식

인터프리터 단계에서는 프로그램을 내부적으로 표현하는 바이트코드들을 순차적으로 읽으며 각 코드에 맞는 기능을 수행한다. 인터프리터는 각 바이트코드마다 해당 기능들을 대응시켜 수행해줘야 하므로 굉장히 느리다. 베이스라인 JIT 단계에서는 이 비용을 없애기 위해 베이스라인 JIT 컴파일러를 사용하여 바이트코드들을 컴파일 한다. 바이트코드마다 실행되는 코드들을 하나로 이어 붙여줄 뿐이다. 베이스라인 JIT의 컴파일러의 목적은 대응하는 비용을 줄이는 것이므로 단순한 번역 수준의 컴파일 만을 하며 최적화를 거의 적용하지 않는다. 마지막으로 최적화 JIT 단계에서는 바이트코드를 컴파일 할 때 전통적인 방법(control flow analysis, constant propagation, common sub-expression, dead code elimination, register allocation) [9]을 포함한 다양한 최적화를 적용하여 더 양질의 코드를 생성한다. 이 방법들을 적용하기 위해 최적화 JIT 컴파일러는 이전 단계들에서의 함수 실행 정보인 profile을 사용한다. 다음 절에서 더 자세히 살펴보겠지만, profile을 사용하여 변수의 type, 객체의 class, 함수의 scope 등과 같이 원래는 동적으로 매번 결정되어야 하는 값들을 컴파일 시간에 고정시켜서 컴파일을 한다. 한번

기계어 코드를 생성한 뒤에 profile에서 벗어나는 상황에 마주할 수도 있는데, 이 때는 베이스라인 단계로 퇴보를 하여 함수를 실행 한다. 이 과정을 OSR Exit이라고 부른다.

## 제 2 절 최적화 JIT 컴파일러와 Profile

자바스크립트 언어는 변수들의 타입(type), 객체들의 모양(class), 함수의 환경(scope) 등이 정적으로 정해지지 않고 그 값들이 매번 확인되어야 하는 굉장히 동적인 언어이다. 이는 정적으로 컴파일 되는 전통적인 언어들과는 다른 특징이다. 따라서 이러한 것들이 미리 정해져야 하는 전통적인 최적화 방법들은 자바스크립트 프로그램에 바로 적용될 수가 없다. 자바스크립트 엔진은 이 문제를 해결하기 위하여 이전 단계 들에서 실행하면서 관찰한 내용들을 수집하여 사용한다. 이러한 관찰 값들을 profile이라고 하는데, 이 내용을 토대로 변수들의 type, 객체들의 class, 함수의 환경 등에 대한 가정을 내려 최적화 JIT 컴파일러에 전달한다. 이 과정은 그림 3에서 볼 수 있듯이, 자바스크립트 프로그램에 다양한 정보들을 삽입하는 것으로 이해할 수 있다. 그림 3에서 위에 있는 프로그램은 원래 자바스크립트의 문법에 맞게 작성된 것이고, 아래의 프로그램은 여기에 profile의 정보들이 녹아 든 모습을 나타낸다. 아래의 프로그램은 이해를 돕기 위해 작성한 것으로 이는 문법에 맞는 자바스크립트 프로그램이 아니다.




[그림 3] 자바스크립트 엔진에서 Profile의 역할

그림 3의 위 쪽 프로그램을 보면 알 수 있듯이, 자바스크립트 함수의 parameter나 지역 변수들에는 그 것들의 type이 적혀있지 않다. 실제로 이들 변수에는 매번 다른 type의 자료들이 저장될 수 있다. 함수 parameter에 해당하는 obj의 경우, obj.p라는 표현 식이 사용되는 것으로 보아 객체임을 추론할 수 있는데, 자바스크립트의 경우 C++이나 Java처럼 객체의 모양, 즉 class가 정해져 있지 않다. 언제든지 새로운 member를 추가하고 삭제할 수가 있다[10]. 이는 자바스크립트가 prototype 기반의 객체 모델을 사용하기 때문이다. 따라서 객체의 member를 접근하기 전에 그 객체의 class가 현재 무엇인지 매번 확인해주어야 하며, member를 접근할 때에도 동적으로 객체 내의 offset을 찾아줘야 한다. 변수 bar의 경우 함수 내부적으로 정의되지 않은 자유 변수인데 자유 변수의 값을 결정하기 위해선 함수의 환경(scope 또는 environment)이 있어야 한다. 하나의 함수는 다양한 환경에서 실행될 수 있으므로 이 또한 동적으로 결정이 되어야 하며, 자유 변수의 값 또한 이를 바탕으로 동적으로 결정이 된다. 여기서 고정을 시키는 것은 지금까지는 이렇게 관찰되었기 때문에 앞으로도 그러할 것이라는 가정을 깔고 가는 것이다. 하지만, profile에서 벗어나는 상황이 컴파일 후에 얼마든지 발생할 수 있기 때문에 앞서 설명한 것처럼 OSR Exit이 일어날 수 있다.

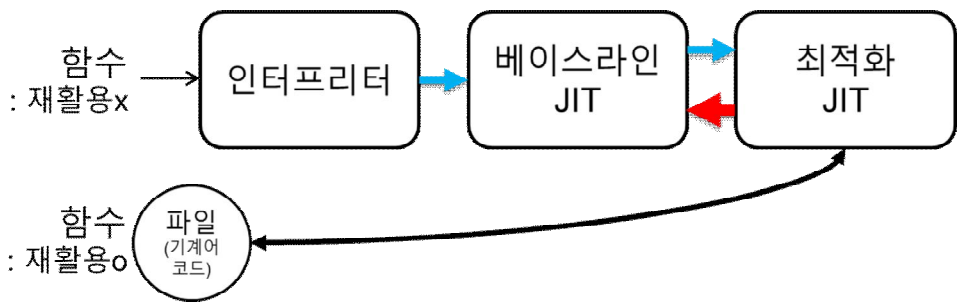
## 제 3 절 WebKit의 JavaScriptCore

이번 연구를 진행하는데 사용한 자바스크립트 엔진은 Apple의 WebKit에 탑재되는 JavaScriptCore(이하 JSC)[8]이다. 이번 절에서는 JSC의 구체적인 구현에 관하여 간략하게 설명한다. JSC는 앞서 설명한 다단계 JIT 컴파일 방식을 취한다. DFG JIT이라는 최적화 JIT 단계를 갖으며 이를 토대로 구현을 진행하였다. JSC는 기본 실행 단위로 전역 프로그램, 프로그램 내의 함수, eval 프로그램의 3가지를 갖는다. 각각에 대해 객체들을 만들어 관리한다. 하나의 실행 단위는 가장 기본적으로 Executable이라는 객체로 표현이 되며 실행에 필요한 대부분의 정보들이 이 객체에 저장된다. 이들은 실행 단계와 관계 없이 실행 단위마다 하나씩 존재한다. 실행 단위에 대한 각종 정보는 CodeBlock 이라는 객체에 저장된다. 최적화 JIT까지 도달한 실행 단위는 OSR Exit의 가능성을 염두에 두어 베이스라인 JIT 단계를 위한 CodeBlock을 계속해서 갖고 있는다.

# 제 3 장 최적화된 기계어 코드의 재활용

## 제 1 절 개요

본 연구에서 제안하는 성능 향상 방식은 이전 실행에서 컴파일된 기계어 코드를 저장하였다가 다음에 프로그램을 실행할 때 이를 재활용하는 방식이다. 이 때의 재활용 대상은 최적화 JIT 컴파일러가 생성하는 기계어 코드이다. 재활용 파일이 없을 때는 기존처럼 인터프리터 단계에서부터 시작을 하지만, 이 함수가 실행을 계속 해나가면서 최적화 JIT 단계에 도달을 해 재활용 파일을 만들게 되면, 그 다음 실행에서부터는 그 파일을 사용하여 최적화 JIT 단계부터 실행을 시작하게 된다.

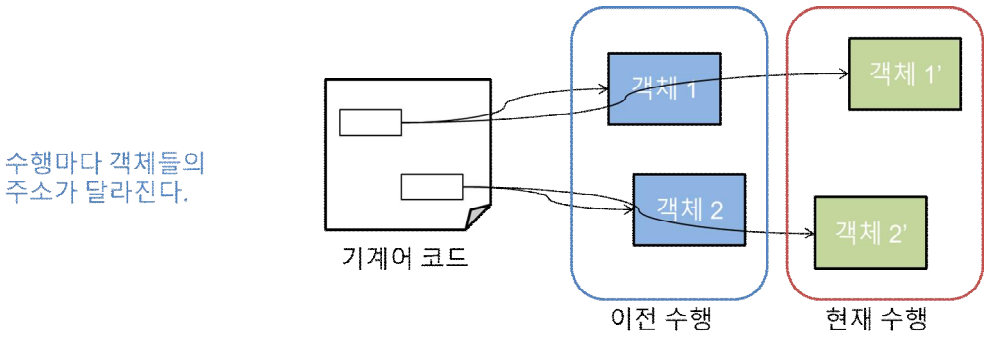



[그림 4] 최적화된 기계어 코드의 재활용 방식

기계어 코드를 재활용하는데 있어서 가장 주된 과제는 코드 내의 포인터들을 현재 수행에 맞게 패치하는 일인데 본 논문에서는 이를 포인터 패칭이라고 한다. 최적화 JIT은 베이스라인 JIT과는 다르게 최적화 기법들이 적용되는데, 이 최적화 과정으로 인해 새로운 포인터들이 다른 포인터들로부터 파생되어 기계어 코드에 삽입된다. 이들을 모두 추적하여 처리해주는 과정이 추가적으로 필요하다. 또한, profile로 인해서 원래는 동적으로 결정되어야 하는 값들이 컴파일 시간에 고정되는 상황이 발생하는데, 이 과정에서 새로운 포인터들이 기계어 코드에 삽입되기

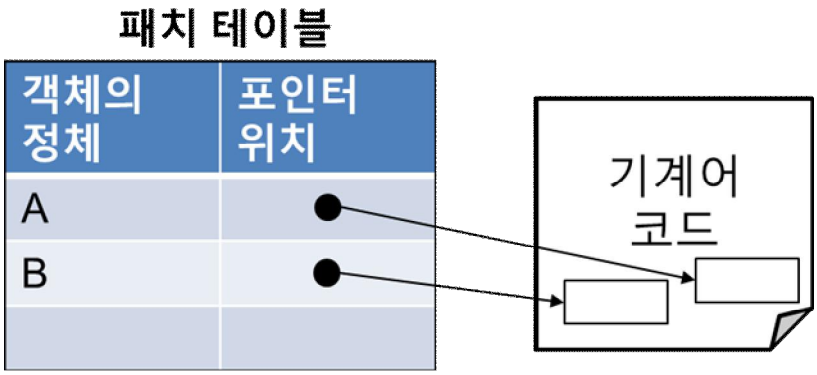
도 한다. 이들은 다른 포인터들과 근본적으로 성질이 달라 일반적인 방식으로는 패치가 어렵다. 따라서 이들을 생성하지 않도록 컴파일러를 고쳐야 한다.

## 제 2 절 포인터 패칭




[그림 5] 기계어 코드 내의 포인터

기계어 코드에는 전역 객체, 라이브러리 객체, 프로그램 내의 string, 엔진 내부 객체, 런타임 함수 등에 대한 다양한 포인터들이 박힌다. 이들은 프로그램의 입력에 상관없이 항상 존재하는 객체들이다. 하지만 메모리 내에서 이들의 위치는 프로그램 실행 마다 바뀔 수가 있다. 따라서 기계어 코드를 재사용하기 위해서는 이 포인터들을 현재의 위치에 맞게 패치 해주어야 한다. 이는 베이스라인 JIT에서도 해결해야 하는 과제이기도 하다.



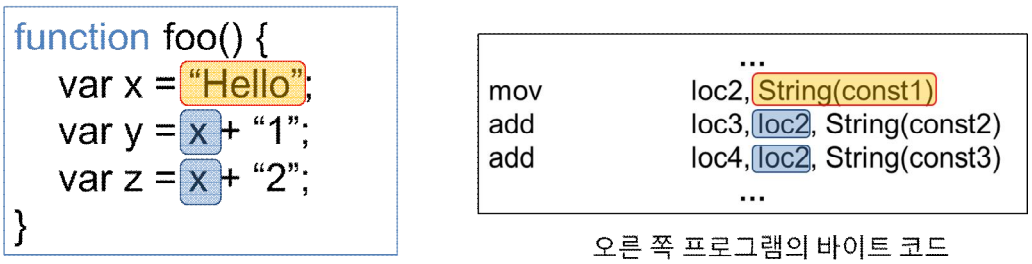

[그림 6] 패치 테이블

이 과제를 해결하기 위해서는 2가지의 정보들이 있어야 한다. 한가지는 기계어 코드 내에 박혀있는 포인터의 위치이며, 다른 한가지는 이 포인터가 가리키는 객체의 정체이다. 기계어 코드가 주어졌을 때 포인터 위치로 어디를 패치해야 하는지를 찾고, 이들을 무엇으로 패치해야 하는지를 객체의 정체를 사용하여 알아낸다. 이 때, 객체의 정체는 내부적으로 다루기 쉽도록 바이트코드 수준에서 표현하여 사용한다. 이러한 정보들을 저장하기 위하여 컴파일러가 그림 6과 같은 테이블을 생성하도록 만든다. 본 논문에서는 이를 패치 테이블이라고 부르겠다. 객체의 정체는 바이트 코드를 parse하는 과정 중에, 포인터의 위치는 컴파일러의 어셈블러가 포인터를 코드에 넣기 직전에 알아낸다. 즉, parse 과정 중에 포인터로 들어갈 객체의 정체들을 queue에 삽입 해놓고, 어셈블러가 포인터를 코드에 넣을 때 queue에서 하나를 빼서 위치와 같이 테이블에 집어 넣는다. 엔진은 기계어 코드를 저장할 때 생성된 테이블을 같이 저장하며, 함수를 실행하기 전에 기계어 코드와 테이블을 불러와 테이블에 적혀 있는 대로 포인터들을 패치한 후 코드를 재사용한다.

## 제 3 절 최적화 과정과 포인터 생성

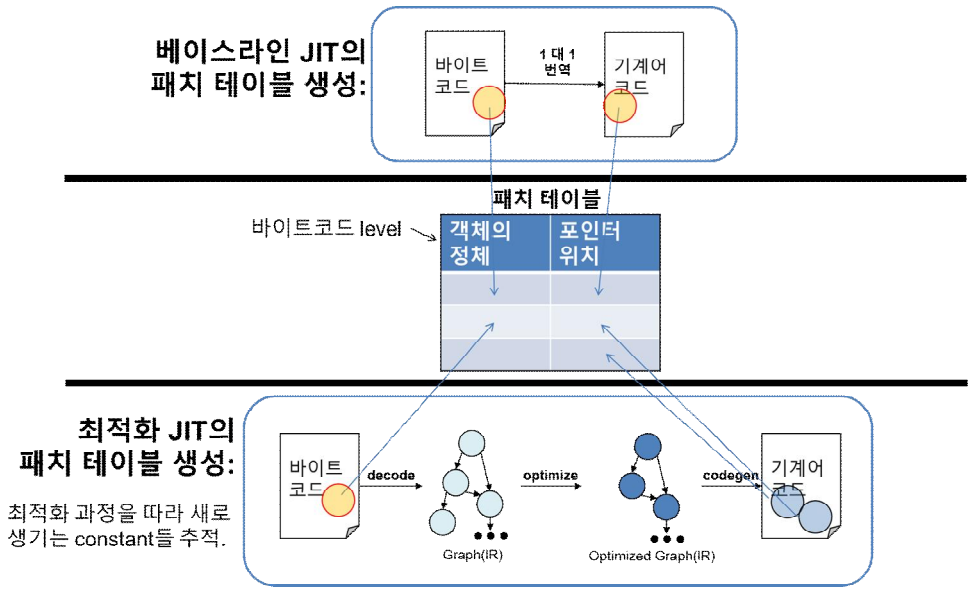
패치 테이블은 베이스라인 JIT에서나 최적화 JIT에서나 생성을

해야 하는 것인데, 이를 생성하는 데에 있어서 최적화 JIT은 2가지의 추가적인 장애물이 있다. 포인터의 위치를 파악하는 것은 어렵지 않으나, 그 것의 정체를 파악하는데 있어서의 어려움이 있다. 본 절과 다음 절은 이에 대해서 살펴볼 것이다. 본 절에서는 최적화 과정 중에 새로 생겨나는 포인터 들의 정체를 파악하는 문제에 대해서 다룬다.




[그림 7] Constant propagation의 예

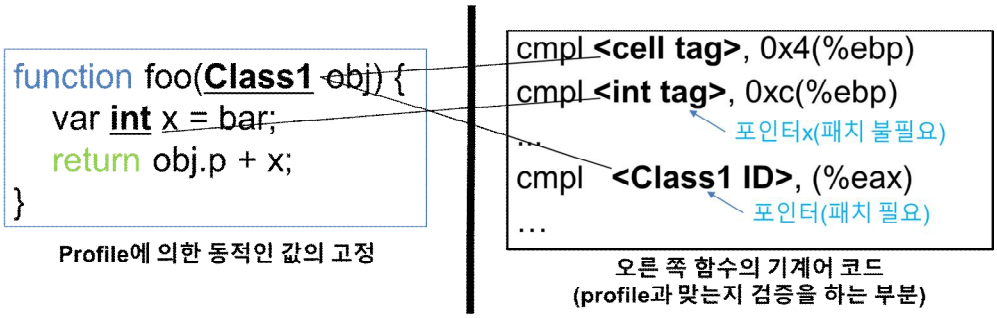
최적화 JIT에서 일어나는 최적화 과정들은 한 곳의 정보를 여러 곳으로 퍼뜨리는 일을 하기도 한다. 이러한 과정에서 하나의 포인터가 여러 곳에 파생적으로 삽입되기도 한다. 그 대표적인 예가 constant propagation[9]이다. Constant propagation은 어떠한 변수가 실제론 constant라는 사실을 이용해 그 변수가 사용되는 곳들을 그 constant 값으로 치환하는 최적화를 말한다. 그 constant가 객체의 포인터인 경우, 하나의 포인터가 원래의 위치뿐만 아니라 다른 곳에도 삽입되는 것이다. 그림 7은 constant propagation의 예를 보여준다. 변수 x는 "Hello"라는 string의 포인터를 값으로 갖는데, x는 그 이후로 변하지 않으므로, x가 사용되는 위치에 x 대신 이 포인터 값이 들어가도 문제가 없다. 그리하여 원래는 첫 줄에 해당하는 바이트코드를 컴파일 할 때만 포인터가 삽입 됐겠지만, x를 사용하는 다른 바이트코드들을 컴파일 할 때에도 포인터가 삽입된다. 따라서, 최적화 JIT에서는 포인터가 가리키는 객체의 정체를 모두 파악하기 위하여 적용되는 최적화 과정들을 모두 추적하여 처리해주는 일이 필요하다. 이러한 점은 베이스라인 JIT과는 매우 다른 점이며 그림 8은 이를 잘 나타내주고 있다.




[그림 8] 베이스라인 JIT과 최적화 JIT에서의 패치 테이블 생성

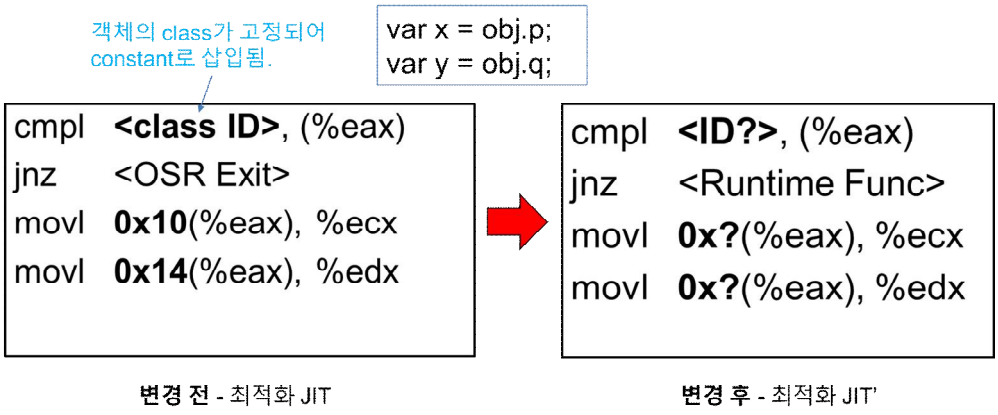
## 제 4 절 Profile과 포인터 생성

앞서 설명하였듯이 profile을 통해 매번 동적으로 결정되어야 하는 값이 컴파일 시간에 정적으로 고정되기도 한다. 이렇게 고정되는 값들 중에서 그 값이 포인터인 경우가 있다. 그림 9에서 obj가 객체(cell)이고 또 그 것의 class가 Class1이라고 고정되면[10], 기계어 코드에 이에 대한 정보가 박힌다. 주로 가정들을 확인하는 guard 코드 부분에 이 정보들이 박힌다. 변수 x도 type이 int라고 고정되면 이 내용도 코드에 박힌다. 변수의 type을 나타내는 tag들은 포인터가 아니기에 패치가 불필요하지만, 포인터로 구현되는 class의 경우 패치가 필요하다.




[그림 9] Profile에 의한 고정의 예

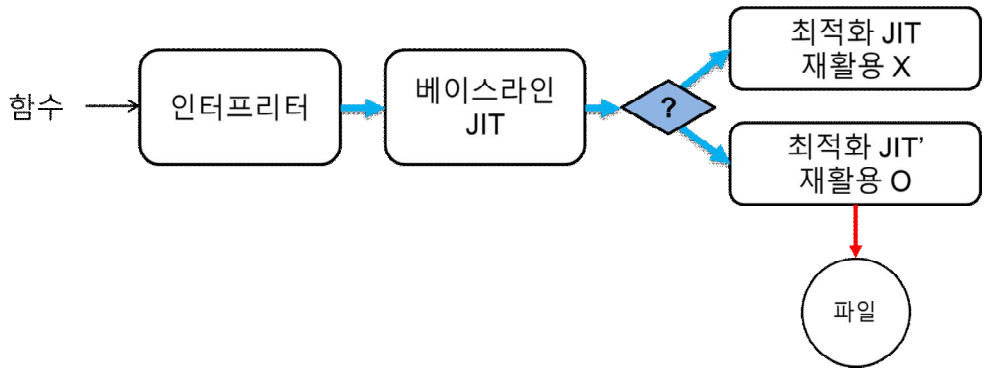
하지만 이러한 방식으로 삽입되는 포인터들은 패치가 매우 어렵다. 이들의 정체가 무엇인지를 결정하기가 어렵기 때문이다. 이들은 이전에 소개된 포인터들과는 달리 프로그램을 관찰하는 것만으로 정체를 알아낼 수가 없다. 이들은 동적으로 관찰된 내용을 바탕으로 정의되기 때문이다. 그림 9의 예에서 Class1은 “함수 foo를 실행하는데 그 argument로 가장 빈번히 관찰된 객체의 class”로 정의가 된다. 이는 프로그램을 실제로 실행 해보기 전 까지는 알 수가 없는 것이다. 전역 객체로 정의되는 객체와 비교를 해보면 이해가 쉽다. 따라서 코드의 재활용이 가능하게 하기 위해선 이들이 처리되어야 한다. 본 연구에서는 이렇게 포인터들을 고정하지 않도록 컴파일러를 변형한다. 즉, 동적으로 결정되어야 하는 값들은 동적으로 처리될 수 있도록 기계어 코드를 생성하게 한다. 이 목표는 베이스라인 JIT의 런타임 메커니즘을 활용하게 하여 달성한다. 그림 10의 예는 이를 잘 보여주고 있다. 변형 전에는 객체의 class가 고정된 채로 코드가 생성되어, 이를 벗어나면 OSR Exit이 일어나도록 되어있지만, 변형 후에는 베이스라인 JIT에서의 런타임 함수를 사용하여, 들어오는 객체의 class가 바뀔 때마다 코드에 박힌 member들의 offset을 바꾸도록 한다.




[그림 10] 컴파일러 변형의 예

## 제 5 절 선택적인 재활용

Profile에 의해 고정을 하는 최적화 외에도 본 연구에서 생략을 해야 했던 많은 최적화들이 있다. 여기에는 method in-lining, array 최적화 등의 최적화가 있다. 이러한 기법들은 이론적으로는 처리가 가능하지만, 포인터의 위치를 찾기 위해서 지금보다 더 복잡한 구현이 필요하다. 결국 변형된 JIT컴파일러(그림 11에서 JIT' )는 원래의 최적화 JIT 컴파일러(그림 11에서 JIT)과 비교했을 때 더 낮은 성능의 기계어 코드를 생성하게 되지만, 컴파일 오버헤드는 완전히 제거가 되고, 여전히 베이스라인 JIT보다는 더 좋은 코드를 생성한다. 변수의 type이 고정되고, common subexpression elimination, dead code elimination, constant propagation, better register allocation 등의 최적화[9]가 여전히 적용되기 때문이다. 따라서 새로운 트레이드오프가 발생하게 된다.




[그림 11] 선택적인 재활용

이에 대한 해결책으로, 본 연구에서는 그림 11에서처럼 사용자가 트레이드오프를 직접 고려해 재활용 여부를 선택하도록 하였다. 재활용을 하는 것이 이득이라고 판단되면 최적화 JIT’ 을 사용해 기계어 코드 저장하도록 했고, 아니라고 판단되면 재활용을 포기하고 기존의 최적화 JIT 사용하도록 하였다.

# 제 4 장 실험 결과 및 분석

## 제 1 절 실험 환경

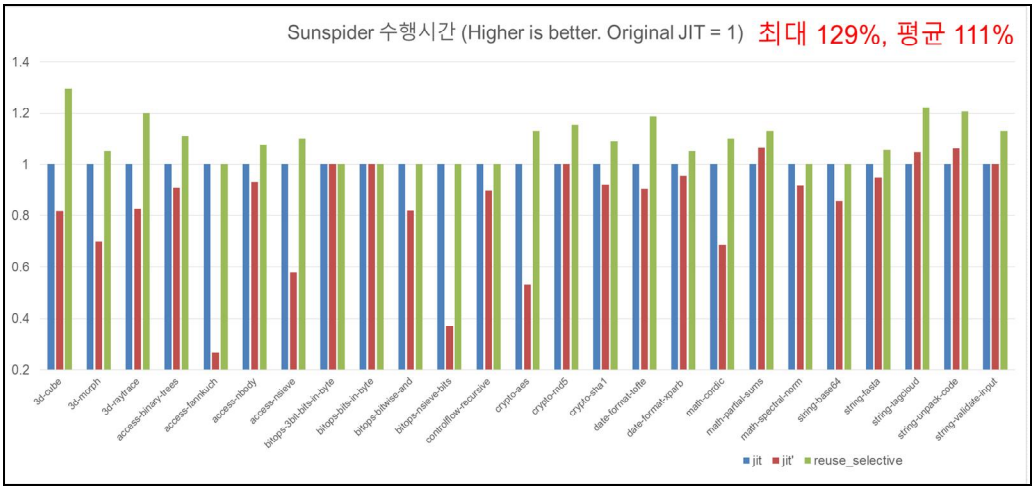
본 논문에서 제시한 방식의 성능 향상을 확인하기 위해 x86의 데스크 탑 환경에서 실험을 진행 하였다. 데스크 탑의 환경은 구체적으로 다음과 같다.

- CPU : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz (8-core)
- RAM : 16GB
- OS : Ubuntu 16.04, Linux kernel 4.2.0-34-generic

자바스크립트 엔진의 경우 Apple사의 WebKit에 들어있는 JavaScriptCore[8]를 사용하였다. WebKitGTK-2.12.0 버전에서 구현을 진행하였다. 구현한 엔진에서 Sunspider 벤치마크[3]를 실행하여 수행 시간을 측정하였다.

## 제 2 절 실험 결과

그림 12은 원래의 엔진(jit, 청색), 변경된 엔진(jit', 적색), 변경된 엔진에서의 선택적 재활용(reuse_selective, 녹색)의 성능을 비교한 결과이다. 숫자가 높을수록 실행 시간이 짧다. 원래의 엔진의 성능을 1로 두었다. 선택적으로 재활용을 한 경우(이상적인 경우), 최대 29%, 평균 11%의 성능 향상이 있음을 확인할 수 있다.




[그림 12] 원래의 엔진(jit), 변경된 엔진(jit’), 변경된 엔진에서의 선택적 재활용(reuse_selective) 성능 비교

## 제 3 절 관찰 결과

재활용의 대상이 되는 함수들 중에서 73%가 성능에 이들을 준다고 판단되어 재활용 되었다. JIT’의 경우 소수의 함수들이 대부분의 성능을 저하시켰는데, 구체적으로 7%의 함수가 전체 성능을 17%로나 저하시켰다. 이들 함수들을 직접 분석한 결과 array 최적화의 생략이 주된 원인으로 추측되고 있다. Array 최적화를 통해 자바스크립트 배열이 C의 배열로 최적화가 됐어야 했으나 그러지 못하고 일반 자바스크립트 객체로 남아있게 된다. 결국 array에 대한 접근이 느려지게 되는데, array에 대한 접근은 loop을 통해 워낙 자주 일어나기 때문에 눈에 띄는 성능의 하락을 가져오게 된다.

# 제 5 장 결론 및 향후 연구

본 연구는 갈수록 증대되는 자바스크립트의 중요성에 맞추어 더욱 향상된 성능의 자바스크립트 엔진을 개발하고자 함이다. 기존의 JIT 컴파일 방식에서 더 나아가 최적화 JIT 컴파일러에서 생성된 코드를 재활용하는 방식을 제안하며 JSC 엔진에서 이를 구현해 보았다. 기존 최적화 JIT 대비 평균 11%, 최대 29%의 성능 향상이 있었다.

향후에는 사용자가 선택을 하는 것이 아니라 컴파일러가 자동으로 재활용 대상을 선택하도록 heuristic을 고안하는 것이 좋을 것이라고 생각한다. 또한, profile로 고정되어 생성되는 포인터들도 지금과 같이 처리하는 것이 아니라, 패치 시점을 뒤로 조금 미루어 실행 정보를 모은 뒤 패치하도록 구현하는 방식에 대해서도 추가적인 연구가 필요하다고 생각한다.

# 참고 문헌


[1] http://mean.io/

[2] Aycock, J. “A brief history of just-in-time”. *ACM Comput. Surv. 35,* 2, 2003

[3] https://webkit.org/perf/sunspider/sunspider.html

[4]오형석, 문수묵. “자바스크립트 적시 컴파일러를 위한 생성 코드 재사용”. *한국컴퓨터종합학술대회 논문집, 한국정보과학회*, 2010

[5]박혁우, 문수묵. “내장형 시스템을 위한 자바스크립트 선택적 선행 컴파일”. *가을 학술발표논문집, 한국정보과학회*, 2012

[6] Dong-Heon Jung, Soo-Mook Moon, and Hyeong-Seok Oh. “Hybrid compilation and optimization for Java-based digital TV platforms”. *ACM Trans. Embedd. Comput. Syst.* 13, 2014

[7] HyukWoo Park, Wonki Jung, Soo-Mook Moon. “JavaScript Ahead-of-Time Compilation for Embedded Web Platform”. *The 13th IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2015

[8] https://webkitgtk.org/

[9] Aho, A., Lam, M., Sethi, R., and Ullman, J. “Compilers: principles, techniques, and tools”, vol. 1009. *Pearson/Addison Wesley*, 2007.

[10] Hölzle, U., Chambers, C., AND Ungar, D. “Optimizing dynamically-typed object-oriented languages with polymorphic inline caches”. *In Proceedings of the ECOOP,* 1991

# Abstract

# Recycling the Optimized Machine Codes Generated by JavaScript Engine

Sung-kook Kim

Dept. of Electrical and Computer Engineering

The Graduate School

Seoul National University

JavaScript is spreading across computers fast and its performance is becoming very important. In this paper, I propose a method of reusing the machine codes, generated by optimizing JIT compilers in multi-tiered JITC engine, to enhance the performance of the JavaScript engines. In this method, machine codes generated by optimizing JIT compiler are permanently stored in a file, and it is reused at the next execution of the program. Thus, compilation overheads are removed from execution, and optimized machine codes are used from the beginning.

To be reused, pointers in the machine codes needs to be patched according to current execution. For this purpose, a table, which records the locations of pointers in the code and the identity of the pointed objects, are stored along with the machine code. In the case of optimizing JIT compilers, pointers generated by the process of optimizations and the pointers introduced by runtime profiles should also be dealt to build a complete table. The former is dealt by tracking down each optimization processes, and the latter is dealt by changing the compiler to make codes dynamically evaluate the values for those pointers. I implemented this method on JavaScriptCore, which is one of most popular JavaScript engine, and obtained maximum of 29% and average of 11% increase in performance, when the user was allowed to reuse only those that made the improvements.

Keywords : JavaScript, Virtual Machine, JIT Compiler, Compilation Overhead, Pointers, Machine codes

Student Number : 2015-22777